# Debugging Mastery Problems

## 1

### Problem description:

Write a function that returns the larger of two numbers.

### Buggy code:

def larger\_number(a, b)  
 if a > b  
 return a  
 else:  
 return b

### One kind of right solution:

def larger\_number(a, b):  
 if a > b:  
 return a  
 else:  
 return b

## 2

### Problem description:

Write a function that checks if a number is even.

### Buggy code:

def is\_even(n):  
 return n % 2 = 0

### One kind of right solution:

def is\_even(n):  
 return n % 2 == 0

## 3

### Problem description:

Write a function that returns the sum of all elements in a list.

### Buggy code:

def sum\_of\_list(lst):  
 return sum(lst

### One kind of right solution:

def sum\_of\_list(lst):  
 return sum(lst)

## 4

### Problem description:

Write a function that returns the average of all elements in a list.

### Buggy code:

def average\_of\_list(lst):  
 return sum(lst) / len(lst

### One kind of right solution:

def average\_of\_list(lst):  
 return sum(lst) / len(lst)

## 5

### Problem description:

Write a function that finds the second largest number in a list.

### Buggy code:

def second\_largest(lst):  
 lst.sort()  
 return lst[1]

### One kind of right solution:

def second\_largest(lst):  
 lst.sort()  
 return lst[-2]

## 6

### Problem description:

Write a function that returns a list without duplicates.

### Buggy code:

def remove\_duplicates(lst):  
 return set(lst)

### One kind of right solution:

def remove\_duplicates(lst):  
 return list(set(lst))

## 7

### Problem description:

Implement a basic Stack class with push and pop methods.

### Buggy code:

class Stack:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def push(self, item):  
 self.items.append(item)  
  
 def pop(self):  
 return self.items.pop(0)

### One kind of right solution:

class Stack:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def push(self, item):  
 self.items.append(item)  
  
 def pop(self):  
 return self.items.pop()

## 8

### Problem description:

Write a function that checks if a given string has balanced parentheses.

### Buggy code:

def is\_balanced(s):  
 stack = []  
 for char in s:  
 if char == '(':  
 stack.append(char)  
 elif char == ')':  
 stack.pop()  
 return not stack

### One kind of right solution:

def is\_balanced(s):  
 stack = []  
 for char in s:  
 if char == '(':  
 stack.append(char)  
 elif char == ')':  
 if not stack:  
 return False  
 stack.pop()  
 return not stack

## 9

### Problem description:

Implement a basic linked list with methods to add and remove nodes.

### Buggy code:

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
class LinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
  
 def add(self, data):  
 new\_node = Node(data)  
 if not self.head:  
 self.head = new\_node  
 else:  
 current = self.head  
 while current.next:  
 current = current.next  
 current = new\_node  
  
 def remove(self, data):  
 if not self.head:  
 return  
 if self.head.data == data:  
 self.head = self.head.next  
 return  
 current = self.head  
 while current.next:  
 if current.next.data == data:  
 current.next = current.next.next  
 return  
 current = current.next

### One kind of right solution:

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
class LinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
  
 def add(self, data):  
 new\_node = Node(data)  
 if not self.head:  
 self.head = new\_node  
 else:  
 current = self.head  
 while current.next:  
 current = current.next  
 current.next = new\_node  
  
 def remove(self, data):  
 if not self.head:  
 return  
 if self.head.data == data:  
 self.head = self.head.next  
 return  
 current = self.head  
 while current.next:  
 if current.next.data == data:  
 current.next = current.next.next  
 return  
 current = current.next

## 10

### Problem description:

Write a function that finds the shortest path in a maze using DFS.

### Buggy code:

def dfs(maze, start, end):  
 stack = [start]  
 visited = set()  
 while stack:  
 current = stack.pop()  
 if current == end:  
 return True  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 stack.append(neighbor)  
 return True  
  
def get\_neighbors(maze, pos):  
 neighbors = []  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 for dx, dy in directions:  
 x, y = pos  
 nx, ny = x + dx, y + dy  
 if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] != 'W':  
 neighbors.append((nx, ny))  
 return neighbors

### One kind of right solution:

def dfs(maze, start, end):  
 stack = [start]  
 visited = set()  
 while stack:  
 current = stack.pop()  
 if current == end:  
 return True  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 stack.append(neighbor)  
 return False  
  
def get\_neighbors(maze, pos):  
 neighbors = []  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 for dx, dy in directions:  
 x, y = pos  
 nx, ny = x + dx, y + dy  
 if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] != 'W':  
 neighbors.append((nx, ny))  
 return neighbors